Flexible Annotations API

Programmer's Guide

Revision 1.0a

May 2001

World Wide Web:

http://developer.intel.com/software/products/opensource/

Table Of Contents

Introduction 1

This document describes application programmer interfaces (APIs) that enable the development of effective code analysis tools for the Intel® Itanium™ architecture. Two key observations motivated this work:

1. Exploitation of architectural features such as predication and speculation require advanced compiler optimization techniques (e.g. if-conversion, trace scheduling, loop unrolling, etc.). Using these features requires a significant number of transformations and a great deal code motion. Without an understanding of what the compiler has done, the analysis task can be very difficult. With the tools provided here, experimental compilers can include annotations in the compiled executable to record the compiler's structural view of the code (e.g., control flow graph, memory dependence information, etc.). Once encoded in a binary, this information can be accessed by analysis tools to help correlate compiler transformations and structures with dynamic information.
2. By using hardware performance monitors, performance simulators, or code instrumentation, it is possible to find the most time consuming places in a program. Once these "hot spots" have been identified, diagnosis of performance issues requires a combination of architectural, microarchitectural, and compilation technology. Performance simulators and hardware performance monitors can produce accurate cycle counts and record important microarchitectural events on a per-IP basis. By using annotations, these IPs can be mapped back to specific functions, basic blocks, instructions or any other structure the compiler wishes to annotate.

Based on these observations, we have developed these APIs to support the development of performance analysis tools beginning with the compiler and ending with report-generating capabilities. As shown in Figure 1, research compilers for the Intel® Itanium® architecture can use the annotation library to encode compiler derived information into the binary. The library consists of:

1. Flexible Compiler Annotation API - used to insert annotations into a binary at compilation time
2. Flexible Annotation Decoder API -- used to extract annotation information from a binary based on IP
3. Annotation Post-Procesing API -- a context and convention sensitive layer to post-process annotations.
4. Example tools to usage model

If combined with a runtime tool (a simulator or hardware performance monitor driver) that records dynamic events (data cache misses, branch mispredictions, etc.) on an IP basis, the annotations allow these events to be correlated with compiler-based information.

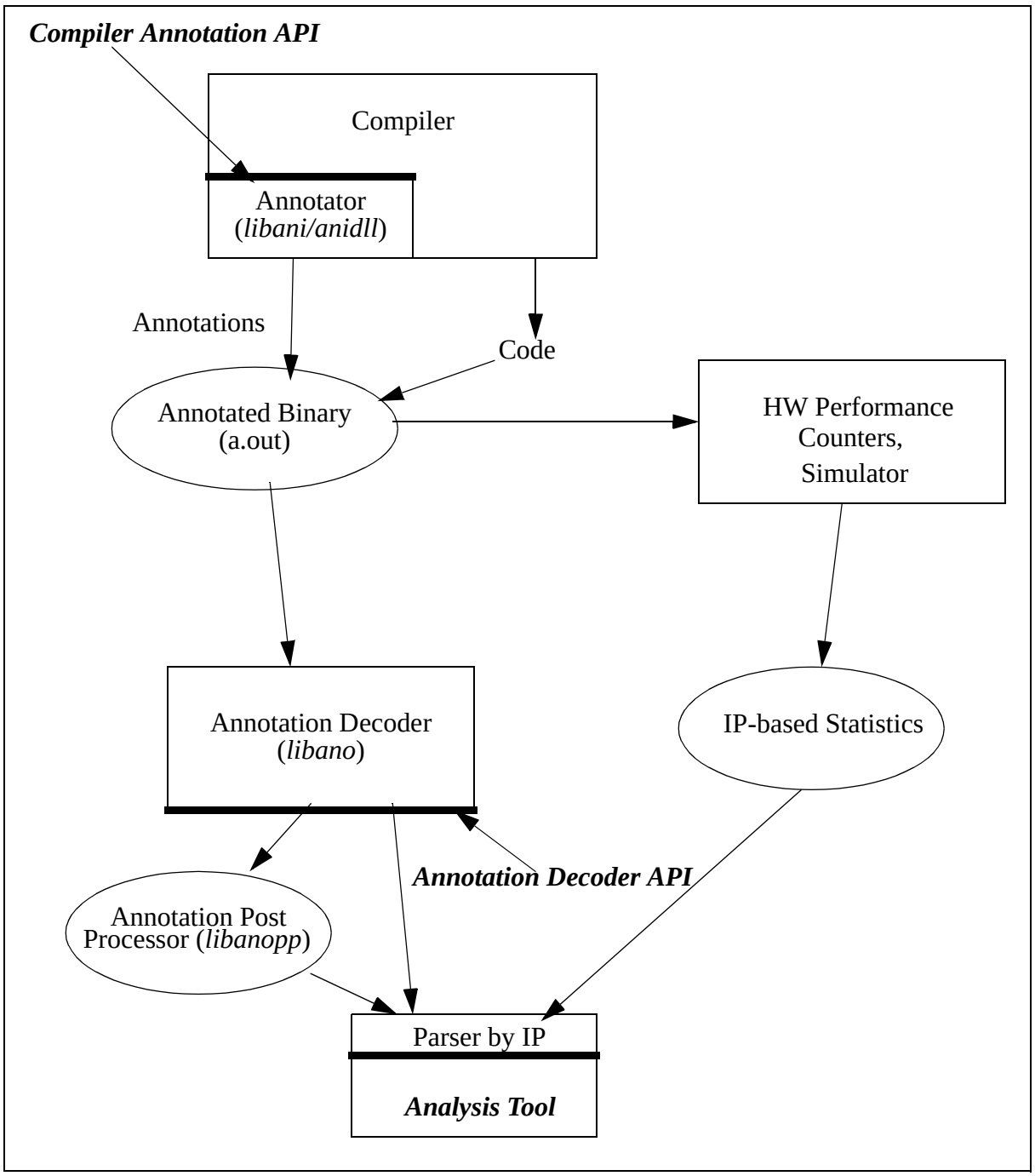

Figure 1-1. Annotation Based Analysis Flow.

This document describes the annotation APIs and a set of suggested base usage conventions. Accordingly the document is structured into four major sections:

1. Flexible Compiler Annotations API - *ani*: describes the mechanisms and programming interface used by the compiler to annotate binaries.
2. Flexible Annotations Decoder API - *ano*: describes the mechanisms and programming interface used by analysis tools to read annotated binaries
3. Annotations Post Processing API - *anopp*: describes the helper functions which, following from the set of conventions, allow for a layer of post processing annotations read from an annotated binary.
4. Annotation Hierarchy Conventions - since the annotations are highly configurable, it is possible to assemble a nearly arbitrary set of annotations. To encourage tool-sharing, we describe a very basic annotation hierarchy. If this hierarchy convention is followed, other tools that also use the hierarchy are likely to work on code generated by a complying compiler. Since the annotations are flexible (any superset of this hierarchy is compatible), it is possible to "hang" new attributes off the existing structure provided that the structure and parent/child relationships haven't changed. This set of conventions provides a basis for writing performance tools using the API, although it is not required to make use of the annotations library.

Flexible Compiler Annotations 2

2.1 Introduction

The following functions are provided to annotate information into and extract annotations from compiled binaries. The annotation libraries allow the compiler to flexibly control the name, type, structure, and amount of information it wishes to embed in a program's binary. This is accomplished by registering the kind of information the compiler wishes to provide about a program at the beginning of the annotation process. Toolsets that later read the compiled binary extract the registered information to know what data is present and its structure.

2.2 Dealing with Contexts

The interface requires the compiler to provide information about a program in a hierarchal manner. For example, the compiler may wish to provide information about several different functions. Within each function, the compiler may wish to provide information about several different basic blocks each with its own set of information. Each position in this hierarchy has its own "context" or set of information. Contexts will have parent contexts (the top will have ROOT as a parent) and may have children contexts (e.g. the basic block context is a child of the function context). This "context tree" provides an organized framework for specifying information based on the position in the hierarchy (or node in the tree).

Each piece of information described in a context is referred to as an "attribute" of the context. For instance, FUNC_NAME may be an attribute of FUNC. The context to which these attributes belong is called the parent context (e.g. FUNC is the parent context of the attribute FUNC_NAME).

Each of the attributes contained in a context will have a "type" associated with it (e.g. FUNC_NAME may be of type "ann_string"). These types indirectly correspond to C types or lists of a particular C type. Consider for example, BB_PRED_NUMS, an attribute of context BB, which refers to the predecessor basic blocks (by number) of a particular basic block. BB_PRED_NUMS may be of type "ann_uint16_list". When an annotation of BB_PRED_NUMS is made, the value annotated is added to a BB's BB_PRED_NUMS list.

In summary, a context refers to a generic container of information that is meant to be associated with a position in an information hierarchy specified by the compiler. An attribute refers to a specific piece of information contained within a context. The attribute has a type which denotes how the information is stored.

2.3 Object Lists And Traversal Of The "Object Tree"

As the compiler performs annotations, objects are created. Each object is an instantiation of a particular context containing values for some or all of the attributes in that context. Attributes of an object that are annotated by the compiler will be emitted into the binary and associated with the attribute's name. A tool reading the program binary may then query attribute names (i.e. FUNC_NAME) of objects and receive the corresponding values annotated by the compiler.

Before the compiler can annotate information about an object, it must be opened. In the current implementation, there can only be one open object at a time. When the compiler performs an *ani* call, the open object receives the information annotated by the compiler. The context which describes this object is referred to as the current "open" context. This attribute must also be the registered identifier for the context. For example, the attribute BB_NUMBER may be registered as the identifier for objects of context type BB. When *ani_open_obj* is annotated by the compiler not the attribute BB_NUMBER, a BB object is opened.

If an *ani_close_obj* is made, the current open object will be closed by resetting the current object to its parent object. For example, if the current open object is of context type BB, it will be closed and its parent object, of context type FUNC, will be reopened for annotation. If the compiler wishes to open another BB object, it can call *ani_open_obj* on the identifying attribute of the BB context (i.e. BB_NUMBER). When the compiler has completed annotating, the *ani_close* call is made, and the annotations are emitted to the assembly file

When an object is created, it is inserted into a list of objects of the same context type. The list to which the object is inserted is an attribute of the current open object. All elements of the list share this object as their parent object. For example, when a BB is created, it is inserted into the list of BB's belonging to the current open object (of context type FUNC). The name of the object list an instantiated context belongs to is an attribute of its parent context. For instance, _BB_LIST may refer to a list of BB's. _BB_LIST will be an attribute of FUNC if FUNC is the parent context of the BB context. In this manner objects are bound together into an "object tree". The compiler is responsible for registering which object list objects of a particular context type belong to. For instance, the compiler will register _BB_LIST as the name of the object list a BB object will belong to within the FUNC context.

Just as the context tree is created through a process of registration, an object tree is created through compiler annotation. By opening and closing objects, the "object tree" can be traversed by the compiler and different nodes of the tree can be annotated. When an object is opened, a step down the object tree is made. The current open object is searched for a list of objects that share the context type of the object being opened. For instance, if the compiler calls *ani_open_obj*, the current open object is searched for the attribute _BB_LIST since _BB_LIST was registered as the name of the list containing BB objects. If the _BB_LIST of the current open object is found, it is traversed to find the BB the compiler wishes to annotate. If the BB isn't found, it is created and attached to the list _BB_LIST. A step down in the object tree is made when the BB is opened and becomes the current open object. When an object is closed, the current open object is reset to the object's parent and a step up the object tree is made.

Different objects of the same context type will be distinguished from each other in an object list by a compiler-designated attribute that uniquely identifies an element of the list. The compiler is responsible for registering which attribute of a context uniquely identifies it from other instances of the context. For instance, BB_NUMBER may be a BB attribute which distinguishes a BB object from other BBs in a particular _BB_LIST.

In summary, when an object is created, it is attached to an object list. Object lists are used to bind objects together into an "object tree". The compiler traverses the object tree through explicit open/close object calls. Objects that belong to a given list are distinguished from each other by a unique identifier which must be an attribute of the context from which the object was created.

2.4 Registration

The compiler creates the context tree and the associated attributes through a process of registration. The following functions have been added to the annotation interface to ensure this functionality.

void ani_reg_ctxt(char *ctxt_name, char *parent_ctxt_name);

This function creates a new context and attaches it to the context tree. The ctxt_name field is used to create a new context container that can be filled with attributes. The new context container will be attached as a child of the context referred to by parent_ctxt_name. parent_ctxt_name must be a context that has already been registered. The first context in tree must have a parent_ctxt_name of ROOT.

```
i.e. ani_reg_ctxt(“BB”, “FUNC”);
```

A BB context container will be created and attached as a child of the FUNC context. The FUNC context must be registered before this call.

A comment describing a context can be added explicitly.

```
i.e. ani_reg_ctxt_comment(“FILE”, “File: Assembly files in binary”);
```

void ani_reg_attr(char *attr_name, char *ctxt_name);

The register_attr function creates a new attribute (called attr_name) and associates it with a registered context by name (ctxt_name). The register_attr function does not create a new attribute entry in ctxt_name. It simply maps which attributes are associated with which contexts.

```
i.e. ani_reg_attr(“BB_NUMBER”, “BB”);
```

Here BB_NUMBER is the name of an attribute. BB_NUMBER is associated with the registered context BB.

A comment describing an attribute can be added explicitly.

```
i.e. ani_reg_attr_comment(“BB_NUMBER”, “BB: Basic Block Number”);
```

void ani_reg_attr_data_type(char *attr_name, ann_enum_types data_type);

ani_reg_attr_data_type specfies an attribute’s data type. That is, when an annotation of attr_name is performed by the compiler, it will pass in “data_type” data. “data_type” must be a data type predefined by the annotation library. Refer to Table 3-1 on page 12 for a list of the data types currently recognized by the annotate library:

```
i.e. ani_reg_attr_data_type(“BB_NUMBER”, ann_uint16);
```

A BB_NUMBER is a value of type ann_uint16.

```
i.e. ani_reg_attr_data_type(“_BB_LIST”, ann_obj_list);
```

A _BB_LIST is of type ann_obj_list

void ani_reg_obj_list(char *list_name, char *ctxt_name, char *attr_id);

The *ani_reg_obj_list* function specifies the name of the object list a “ctxt_name” object should be attached to when it is created. list_name must be an attribute of “ctxt_name”’s parent. The attr_id is the attribute of a “ctxt_name” object that distinguishes it from another “ctxt_name” objects. The identifier only distinguishes an object from other objects in the parent object’s “list_name”. This implies that objects may have the same id value as long as they belong to different parent objects.

```
i.e. ani_reg_obj_list(“_BB_LIST”, “BB”, “BB_NUMBER”);
```

This call specifies that BB objects are contained in a _BB_LIST, an object list attribute of BB’s parent context. BBs in a _BB_LIST will be distinguished from each other by their BB_NUMBER value. BB_NUMBER is a registered attribute of a BB. If the compiler calls *ani_open_obj*, the object’s BB_NUMBER value must be passed in to determine whether the requested BB exists or needs to be created and added to the parent’s _BB_LIST.

void ani_reg_xiplabel(char *xip_lbl_attr, char *xip_attr);

In order to map information annotated by the annotation libraries to a simulation run, it is necessary to match instruction addresses. However, the compiler has no control of the particular XIP an instruction will be given by the linker. In order to keep track of an xip, a string attribute may be printed as a label in the assembly file. The label can be tracked in the binary through the linker’s relocation process and reassociated with an object’s attribute value once the binary is read. Two calls are provided in the flexible annotation interface to facilitate this XIP tracking mechanism.

An attribute may be registered as a label producer. When the attribute is annotated it will emit a label in the binary. The label will be tracked when the linker replaces it by an XIP. The XIP read from the binary will then be stored in the registered xip_attr field of the same object.

i.e.

```
ani_reg_xiplabel(“FUNC_NAME”, “FUNC_XIP”);
ani_annot_obj(“FUNC_NAME”, “foo”);
```

For instance, these two calls will produce a label “foo” in the assembly file which will be replaced by an XIP thanks to the linker. This XIP will be read from the binary and read into the FUNC_XIP field of the same object that has “foo” for a FUNC_NAME.

void ani_reg_xipoffset(char *xip_off_attr, char *xip_prod_attr, char *xip_attr);

The second call provided to track XIPs is actually used to create an new XIP based off an XIP read from the binary. An attribute may be registered as an XIP offset. When the attribute (xip_off_attr) is annotated and its value is read from the binary it is added to the XIP value stored in the parent’s xip_prod_attr field. The resultant XIP will then be stored in the registered xip_attr field of the same object.

i.e

```
ani_reg_ctxt(“BB”,”FUNC”);.
ani_reg_xiplabel(“FUNC_NAME”, “FUNC_XIP”);
ani_reg_xipoffset(“BB_INSTR_SLOT_OFFSET”, “FUNC_XIP”, “BB_XIP”);
ani_annot_obj(“FUNC_NAME”, “foo”);
...
ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, “5”);
```

The annotation of FUNC_NAME will produce a label “foo” in the assembly file which will produce an XIP read out of the binary and stored in FUNC_XIP. When the binary is read by the tool, the offset “5” will be added to FUNC_XIP and stored in the BB_XIP field of the same object that has 5 for a BB_INSTR_SLOT_OFFSET value. Thus, a BB_XIP is “created” during the time the tool reads the binary. Note that this BB will be a child of the FUNC object that has “foo” for its FUNC_NAME.

2.5 General Notes/Restrictions Regarding Flexible Annotations

1. Any list (be it a ann_obj_list, ann_uint16_list, what have you) will have an implicit attribute created to denote the number of elements which is the attribute with a “#” preceding it.

i.e.

```
ani_reg_attr(“BB_PRED”, “BB”, NULL, false);
ani_reg_attr_value_type(“BB_PRED”, “ann_uint16_list”);
```

The attribute “#BB_PRED” will automatically be created to denote the number of elements in the list BB_PRED.

2. Objects may be opened and closed as the compiler sees fit given that the compiler is at the correct position in the context hierarchy. The compiler need not annotate all attributes of an object before moving onto the next one. The compiler could annotate some of the instructions in a particular basic block. Close the first basic block, annotate some instructions in another basic block. Close it, reopen the first basic block and add additional annotations.
3. New contexts may be added horizontally. In this way, a given parent context may have more than one child context.
4. Only one object may have ROOT as a parent.
5. Number of attributes per context currently limited to MAX_UCHAR (255). No more than MAX_USHORT attributes or contexts may be registered!
6. If a particular object is annotated more than once with the same attribute, and the attribute is a list, the value is simply attached to the list else the value (even strings) or simply replaced!

Flexible Compiler Annotation API - ani 3

This section describes the application programming interface used by the compiler to annotate the code it generates. The compiler calls the annotation routines while emitting assembly code. The annotation library is charged with emitting the appropriate mixture of section encoding, assembler directives into the assembly file.

3.1 Setting Up And Annotating

Annotations are defined by attribute/value combinations. This allows the API to consist of only a few functions. Each function specifies the file pointer of the assembly output file, the name of the attribute and an attribute dependent number of long integer values or a character string.

The attribute names chosen by convention are defined in *Section 6 “Annotation Conventions”* .

void ani_init()

purpose: Initialize structures for the annotation library
input variables: None
note(s): This function must be called before any other annotation calls are made including registration.

void ani_close(FILE *fp)

purpose: Allows the compiler to inform the annotation library when its completed its annotation calls. When this call occurs, the emit routine will be called.
input variables: *fp* - pointer to the assembly file the annotations should be output
note(s): This function must be called once all annotations are complete.

obj_t *ani_open_obj(char* id_attribute, void *val, obj_t *parent_objp)

purpose: If there exists an object of the appropriate context under *parent_objp,* with *val* for its *id_attribute* return a pointer to the object; else create a new object by annotating *val* as its identifier, open the object and return a pointer to it.
returns: pointer to the open object
input variables: *attribute* - registered attribute that has a registered data type to be annotated
parent_objp - pointer to the parent of the object to be opened.
val - value to be associated with attribute.
note(s): When opening the first level in the hierarchy, (context has ROOT for its parent context), *parent_objp* must be NULL.

obj_t *ani_close_obj(obj_t *open_objp)

purpose: Closes an object, returning to the parent.
returns: pointer to the parent of the object just closed
input variables: *open_objp* - pointer to the object that the annotator wishes closed.

void ani_annot_obj (char* attribute, void* val, obj_t *open_objp)

purpose: Annotates *attribute* with the value of *val* interpreted according to the registered data type of *attribute*.
input variables: *attribute* - registered attribute to be annotated.
open_objp - pointer to the open object whose *attribute* is to be annotated.
val - value to be associated with *attribute*.

void ani_del_annotate (char* attribute, obj_t *open_objp)

purpose: Deletes any annotation made of *attribute*.
input variables: *attribute* - registered attribute to be deleted of any annotation.
open_objp - pointer to the open object whose *attribute* is to be deleted.
note(s): This will fully delete the requested annotation. If the annotation is an object list, all objects and subobjects in the list will be deleted.

3.2 Data Registration

void ani_reg_ctxt(char *ctxt_name, char *parent_ctxt_name)

purpose: Create a context container and associates it with its parent context. Calls to this function build the context tree.

input variables: *ctxt_name* - name of the context to be added to the context tree.
parent_ctxt_name - name of the parent context to whom the new ctxt_name container will be attached as a child context.

note(s): 1) The first registered context must be ROOT with NULL for the parent_ctxt_name.
2) The second registered context must have ROOT for its *parent_ctxt_name*.
3) With the exception of #1, *parent_ctxt_name* must be a registered context.
4) Only one context may have ROOT as its parent context.

void ani_reg_ctxt_comment(char *ctxt_name, char *comment)

purpose: Add a descriptive comment to a context.

input variables: *ctxt_name* - name of the context to be added to the context tree.
comment - comment string

note(s): 1*ctxt_name* must refer to a registered context

void ani_reg_attr(char *attr_name, char *ctxt_name)

purpose: Create a new context attribute that may annotated by the compiler.

input variables: *attr_name* - name of the new attribute
ctxt_name - name of the context this attribute is associated with

note(s): 1) *ctxt_name* must be a registered context.
2) *attr_name* must not be a registered attribute.
3) *attr_name* must be the identifying attribute of *ctxt_name*

void ani_reg_attr_comment(char *attr_name, char *comment)

purpose: Add a descriptive comment to an attribute.
input variables: *attr_name* - name of the context to be added to the context tree.
comment - comment string
note(s): *attr_name* must refer to a registered attribute

void ani_reg_attr_data_type(char *attr_name, ann_enum_types data_type)

purpose: For a given attribute, set its data type
input variables: *attr_name* - name of the attribute
data_type - the type of data associated with the given attribute

Table 3-1. List of data types recognized by the annotation library:

Enum Type	Description
ann_bool	Boolean value
ann_int8	Signed 8-bit quantity (char)
ann_uint8	Unsigned 8-bit quantity (char)
ann_int16	Signed 16-bit quantity (short)
ann_uint16	Unsigned 16-bit quantity (short)
ann_int32	Signed 32-bit quantity (int)
ann_uint32	Unsigned 32-bit quantity (int)
ann_flt32	32-bit floating point quantity
ann_flt64	64-bit floating point quantity
ann_string	String (char *), must be NULL terminated
ann_string_list	List of strings
ann_obj_list	List of objects
ann_int8_list	List of signed 8-bit quantities (char)
ann_uint8_list	List of unsigned 8-bit quantities (char)
ann_int16_list	List of signed 16-bit quantities (short)
ann_uint16_list	List of unsigned 16-bit quantities (short)
ann_int32_list	List of signed 32-bit quantities (int)
ann_uint32_list	List of unsigned 32-bit quantities (int)
ann_flt32_list	List of 32-bit floating point quantities
ann_flt64_list	List of 64-bit floating point quantities
ann_xip	Unsigned 64bit integer representing an XIP address

example:

```
ani_reg_attr_data_type("BB_NUMBER", ann_uint16);
```

```
ani_reg_attr_data_type("_BB_LIST", ann_obj_list);
ani_reg_attr_data_type("FUNC_NAME", ann_string);
```

note(s): 1) *attr_name* must be a registered attribute.
2) *data_type* must belong to the current list of recognized data types.
3) An attribute that refers to a list of objects must be of type obj_list.
4) The registered attribute/data_type pair is added to the context with which *attr_name* is associated will be emitted into the binary. These will be the attributes that can be referenced by the ano library.
5) *attr_name* must not already have a registered data_type.

void ani_reg_obj_list(char *list_name, char *ctxt_name, char *id_attr)

purpose: Denotes the name of the list to which an object of context type *ctxt_name* should be attached in the parent object. Also specifies which attribute of the context (*ctxt_name*) uniquely identifies instances in the same list (*list_name*).

input variables: *list_name* - name of obj_list attribute in the parent context of the context referred to by ctxt_name
ctxt_name - the name of the context
id_attr - attribute of *ctxt_name* which distinguishes one object of type *ctxt_name* from another in a *list_name*

example:

```
ani_reg_attr("_BB_LIST", "FUNC", False);
```
_BB_LIST is registered as an attribute of FUNC

```
ani_reg_attr("BB_NUMBER", "BB", False);
```
BB_NUMBER is registered as an attribute of BB.

```
ani_reg_attr_data_type("BB_NUMBER", ann_uint16);
ani_reg_obj_list("_BB_LIST", "BB", "BB_NUMBER");
```

_BB_LIST is the name of the attribute in a FUNC object where BB objects will now be placed. The BB objects in a particular _BB_LIST can be differentiated from each other by their BB_NUMBER values.

note(s): 1) The *list_name* must be a registered attribute.
2) The *ctxt_name* must be a registered context.
3) The *attr_id* must be a registered attribute of *ctxt_name*.
4) *attr_id* must have a registered data type.
5) *attr_id*'s data type cannot be a list, or a bool.
6) *list_name* must be an attribute of the registered parent context of *ctxt_name*.
7) *list_name* must have obj_list as its registered data type.

void ani_reg_xiplabel(char *xip_lbl_attr, char *xip_attr);

purpose: Register an attribute that, when annotated, produces a label in the assembly file.

input variables: *xip_lbl_attr* - attribute that, when annotated, produces a label
2) *xip_attr* - XIP attribute that stores the XIP produced by the linker's replacement of the emitted label.

note(s): 1) *xip_lbl_attr* and *xip_attr* must be registered attributes. 2) *xip_lbl_attr* and *xip_attr* must belong to the same context.
3) *xip_attr* must have a registered "xip" data type.
4) *xip_lbl_attr* must have a registered "ann_string" data type.

void ani_reg_xipoffset(char *xip_off_attr, char *xip_prod_attr, char *xip_attr);

purpose: Register an attribute that is used to calculate an XIP as an offset from a label.

input variables: *xip_off_attr* - value attribute that, when added to the XIP produced by *xip_prod_attr* produces a new XIP.
xip_prod_attr - attribute that, when annotated, produces a label.
xip_attr- XIP attribute that stores the XIP produced by the linker's replacement of the emitted label with the added offset.

note(s): 1) *xip_attr*, *xip_off_attr* and *xip_prod_attr* must be registered attributes.
2) *xip_attr* and *xip_off_attr* must belong to the same context.
3) *xip_attr* must have a registered "xip" data type.
4) *xip_prod_attr* must be a registered label producing attribute.
5) *xip_prod_attr* must belong to the parent context of the context that contains *xip_attr* and *xip_off_attr*.
6) *xip_off_attr* must have a registered value data type. (Can't be a string, list, or bool)

3.3 *ani* API Routines Grouped by Functionality

3.3.1 Annotation

3.3.2 Registration

Flexible Annotations Decoder API - ano 4

4.1 Class Token Operations

char* ano_get_class_name(class_token *obj)

purpose: Print the name of the object class represented by the token.
returns: A pointer to a string with the name of the class.
input variables: *obj* - The *class_token* or *object_token* to get the name of. An item in the class need not have been selected prior to using this routine.

class_token *ano_get_class_token(char *name, class_token *obj)

purpose: Prepare an object token to be instantiated by a later function call. The token is not usable until is actually instantiated. This function must be used before the token can represent a particular member (e.g. basic block **14** of function **foo**) of a class (e.g. all basic blocks).
returns: A class token or NULL if error
input variables: *name* - The name of the object class
obj - Pointer to the new object token to be prepared.
note(s): Object tokens have two parts: a class and an item. This routine only sets the class.

4.2 Class and Attribute Query Operations

char *ano_get_first_class_name(ano_list_token *list)

purpose: Retrieve the first class name from the list of annotated classes.
returns: A pointer to a string with the name of the first class in the class list.
input variables: *list* - List token used to traverse the class list. It will be initialized by using this routine.

char *ano_get_next_class_name(ano_list_token *list)

purpose: Retrieve the next class name from the list of annotated classes.
returns: A pointer to a string with the name of the next class in the class list.
input variables: *list* - List token used to traverse the class list. The list token should reference a previous element in the list.

char *ano_get_first_attr_name(ano_list_token *list, char *class_name)

purpose: Retrieve the first attribute name from the list of annotated attributes belonging to the given class (*class_name*).
returns: A pointer to a string with the name of the first attribute in the list of attributes belonging to the class referred to by *class_name*.

input variables: *list*- List token used to traverse the class list. It will be initialized by using this routine.
class_name - the name of the class which is being queried for its attributes. If *class_name* is NULL, the global list of attributes will be traversed.

char *ano_get_next_attr_name(ano_list_token *list, char *class_name)

purpose: Retrieve the next attribute name from the list of attributes belonging to the given class (*class_name*).
returns: A pointer to a string with the name of the next attribute in the list of attributes belonging to the class referred to by class_name.
input variables: *list* - List token used to traverse the class list. The list token should reference another a previous element in the list.
class_name - the name of the class which is being queried for its attributes. If *class_name* is NULL, the global list of attributes will be traversed.

4.3 Attribute Token Operations

attr_token *ano_get_attr_token(char *name)

purpose: Make a new attribute token for the attribute. Possible attribute names are given in tables in the *ano_get_XXX_attr*() function descriptions below.
returns: A pointer to the attribute token or NULL if an error occurred.
input variables: *name* - The name of the attribute desired.

char *ano_get_attr_name(attr_token attr)

purpose: Retrieve the name of the attribute represented by the token.
returns: Pointer to a string with the name of the attribute.
input variables: *attr* - The attribute token to get the name of.

ano_built_in_types ano_get_attr_type(attr_token attr)

purpose: Retrieve the type of an attribute (the ano version).
returns: An integer which indicates the type according to the following table:

Value	Attribute type
ano_notype	invalid attribute
ano_string	string
ano_uint32	32-bit unsigned integer
ano_uint64	64-bit address (U64)
ano_flt32	single precision floating point
ano_flt64	double precision floating point
ano_str_array	array of strings
ano_uint32_array	array of 32-bit unsigned integers
ano_uint64_array	array of address (U64)

Value	Attribute type
ano_flt32_array	array of single precision floating point numbers
ano_flt64_array	array of double precision floating point numbers
ano_subobj_array	array of objects

input variables: *attr* - An attribute token.

char *ano_get_attr_comment(attr_token attr)

purpose: Retrieve the annotated comment of the attribute represented by the token.
returns: Pointer to a string with the attribute's comment.
input variables: *attr* - The attribute token to get the comment of.

char *ano_get_attr_class_name(attr_token attr)

purpose: Retrieve the name of the class a given attribute belongs to.
returns: Pointer to a string with the name of the class.
input variables: *attr* - The attribute token to get the class name it belongs to.

char *ano_get_subobject_class_name(attr_token subobjtok)

purpose: Retrieve the name a given subobject's class.
returns: Pointer to a string with the name of the class.
input variables: *attr* - The attribute token to get the class name of.
note(s): To retrieve the class name the sub-object attribute belongs to, *ano_get_attr_class_name* should be used. This call is used to retrieve the name of the class the objects in a subobject array belong to. For instance, the elements in a _FUNC_BB_LIST may belong to the class

_BB_LIST. _FUNC_BB_LIST is the subobject attribute array belonging to the class _FUNC_LIST.

int ano_is_valid_attrp(class_token *obj, attr_token attr)

purpose: Test if the specified class owns a particular attribute.
returns: 0 if obj does not have an attribute attr, otherwise non 0.
input variables: *obj* - The class to check for the given attribute. A particular object in this class need not have been selected prior to calling this routine. An *object_token* may also be used for this variable.
attr - A valid attribute token to check for membership in the object class.

4.4 Object Token Operations

object_token * ano_pick_object(class_token *obj, U64 xip)

purpose: Instantiate the object token *obj* with the item at the given "XIP" address. Note that *obj* must have been properly prepared by *ano_get_class_token()* for this function to succeed.
returns: The *object_token* pointer *obj*. This *class_token* will be transformed from a *class_token* to an *object_token*. If an object token is the input, it will be reinstantiated according to the new "XIP". Returns NULL if no object of the correct class is found at *xip*.
input variables: *obj* - The class_token in which the desired object can be found.
xip - The address at which the desired object resides.

object_token *ano_first_object(class_token *obj)

purpose: Instantiate the object token *obj* with the first item in the global static list. *ano_get_class_token*() specifies which list this function should draw from. In general, since the compiler annotates things in increasing "XIP" order, traversal of the static list will return items in increasing "XIP" order.
returns: An *object_token* which points to the transformed input variable *obj*.
input variables: *obj* - A class token which has already be assigned a class with *ano_get_class_token()*.

object_token *ano_next_object(object_token *obj)

purpose: Change the object token to point to the next object in the class. Use in combination with *ano_first_object()*.
returns: A pointer to the input variable *obj*.
input variables: obj - An object token instantiated with *ano_first_object()*.
note(s): The *object_token obj* should already be pointing to an item in the object class. This could either be done with *ano_first_object()* or *ano_pick_object()*.

int ano_is_valid_objectp(object_token *obj)

purpose: Test if the object has a valid class and points to a valid item in the class.
returns: 0 if *obj* points to an invalid item in its class, otherwise it returns non 0.
input variables: *obj* - The object to test.
note(s): This routine works well as a loop termination test when used with *ano_first_object()* and *ano_next_object()*.

char *ano_get_object_comment(object_token *obj)

purpose: Retrieve the annotated comment of the object represented by the token.
returns: Pointer to a string with the object's comment.
input variables: *attr* - The object token to get the comment of.

4.5 Run Token Operations

int ano_read_binary(char *filename)

purpose: Read static annotations from a binary file and create internal data structures.
returns: 0 if successful. 1 if not.
input variables: *filename* - A string with the name of the file to read.

4.6 Data Retrieval

uint32 ano_get_uint32_attr(object_token *obj, attr_token attr, int* valid)

purpose: Retrieve the value of an unsigned 32-bit integer (or smalle integer) attribute from a particular object.
returns: uint32 value of the attribute (*attr*) for a given object (*obj*). *valid* is set to 1 if *attr* wasn't annotated for *obj*.
input variables: *obj* - An instantiated object token.
attr - The desired attribute (which is legal for *obj*'s class).
valid - Pointer to valid flag.

flt32 ano_get_flt32_attr(object_token *obj, attr_token attr, int *valid)

purpose: Retrieve the value of a single precision floating point attribute from a particular object.
returns: flt32 value of the attribute (*attr*) for given object (*obj*). *valid* is set to 1 if *attr* wasn't annotated for *obj*.
input variables: *obj* - An instantiated object token.
attr - The desired attribute (which is legal for *obj*'s class).
valid - Pointer to valid flag.

flt64 ano_get_flt64_attr(object_token *obj, attr_token attr, int* valid)

purpose: Retrieve the value of a double precision floating point attribute from a particular object.
returns: fl64 value of the attribute (*attr*) for given object (*obj*). *valid* is set to 1 if *attr* wasn't annotated for *obj*.
input variables: *obj* - An instantiated object token.
attr - The desired attribute (which is legal for *obj*'s class).
valid - Pointer to valid flag.

char *ano_get_str_attr(object_token *obj, attr_token attr, int* valid)

purpose: Retrieve the value of a string attribute from an object

returns: String value of the attribute (*attr*) for given object (*obj*). *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - An instantiated object token
attr - The desired attribute (which is legal for *obj*'s class).

U64 ano_get_U64_attr(object_token *obj, attr_token attr, int* valid

purpose: Retrieve the value of a U64 (64 bit unsigned integer) attribute from an object

returns: U64 value of the attribute (*attr*) for given object (*obj*). *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - An instantiated object token
attr - The desired attribute (which is legal for *obj*'s class).
valid - Pointer to valid flag.

uint32 ano_get_uint32_array_attr(object_token *obj, attr_token attr, int index, int *valid)

purpose: Retrieve the value of one 32-bit unsigned integer item in an attribute array.

returns: uint32 value stored in the array (*attr)* of *obj* at the given *index*. *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - A selected object in a class.
attr - An attribute which contains a array of values, rather than a single value.
index - The index of the item in the array. The first item in the array has an index of 0.
valid - Pointer to valid flag.

flt32 ano_get_flt32_array_attr(object_token *obj, attr_token attr, int index, int* valid)

purpose: Retrieve the value of one 32-bit (single precision) floating point item in an attribute array.

returns: flt32 value stored in the array (*attr)* of *obj* at the given *index*. *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - A selected object in a class.
attr - An attribute which contains a array of values, rather than a single value.
index - The index of the item in the array. The first item in the array has an index of 0.
valid - Pointer to valid flag.

flt64 ano_get_flt64_array_attr(object_token *obj, attr_token attr, int index, int* valid)

purpose: Retrieve the value of one 64bit (double precision) floating point item in an attribute array.

returns: flt64 value stored in the array (*attr)* of *obj* at the given *index*. *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - A selected object in a class.

attr - An attribute which contains a array of values, rather than a single value.
index - The index of the item in the array. The first item in the array has an index of 0.
valid - Pointer to valid flag.

char *ano_get_str_array_attr(object_token *obj, attr_token attr, int index, int *valid)

purpose: Retrieve the value of one string item in an attribute array.

returns: string value stored in the array (*attr)* of *obj* at the given *index*. *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - A selected object in a class.
attr - An attribute which contains a array of values, rather than a single value.
index - The index of the item in the array. The first item in the array has an index of 0.
valid - Pointer to valid flag.

void ano_get_object_array_attr(object_token *obj, attr_token attr, int index, object_token *sub_object, int *valid)

purpose: Retrieve the *sub_object* which points to one item in an attribute array.

returns: Sets *sub_object* to point to the object stored in the array (*attr*) of *obj* at the given index. *valid* is set to 1 if *attr* wasn't annotated for *obj*.

input variables: *obj* - A selected object in a class.
attr - An attribute which contains a array of subobjects, rather than a single sub-object.
index - The index of the item in the array. The first item in the array has an index of 0.
valid - Pointer to valid flag
sub_object - Token to be given the proper information to access the sub-object.

void ano_get_parent_object(object_token *obj, object_token *parent_obj)

purpose: Instantiate the object token *parent_obj* with the object that is a direct parent to *obj* in the object tree.

note(s): *obj* must have been properly prepared by *ano_get_class_token()* for this function to succeed. *parent_obj* will receive the class token appropriate to *obj*'s parent. In this way, the object tree may be traversed in both directions (towards the root as well as towards the children)

returns: NONE

input variables: *obj* - A pointer to the child's object_token.
parent_obj -Instantiated object token to be filled in by the function.

4.7 ANO API Routines Grouped by Functionality

4.7.1 Token Gathering Functions

4.7.2 Token Queries

4.7.3 Class and Attribute Queries

4.7.4 Data Structure Manipulation

4.7.5 Token Instantiation Routines: Picking Items

4.7.6 Validity Test

4.7.7 Data Access Routines

Annotations Post Processing API - anopp 5

5.1 Introduction

Part of the purpose of the flexible annotations is to hide the implementation details from the user of the annotations library. Therefore, the annotations are allowed no “knowledge” of the underlying structure of annotated data. The registration of the context hierarchy is the only means of communication to the structure of the data. In other words, the library has no understanding of how a BB object relates to a FUNC object other than the BB is registered as a child to FUNC. Any layer of functionality on top of the *ano* library which “understands” the chosen conventions (See “Annotation Conventions” on page -26.) must be written into the *anopp* library.

5.2 Annotations Post Processing Functions:

void create_bb_xips()

purpose: To associate an XIP with each BB object. Reads the list of BB objects associated with each label and creates a BB_XIP by adding the LABEL_XIP to the BB_INSTR_SLOT_OFFSET of the corresponding BB object with the BB_NUM field referred to in the LABEL_BBNUM_LIST.

returns: NONE

input variables: NONE

note(s): Since the annotations conventions allow for a BB_XIP to be based off of a label which is not a parent to the bb, the creation of a BB_XIP cannot be registered. The registration of BB_XIP is simply to open up space for the post processor to fill in. Therefore the BB_XIP must be created after the binary is read (*ano_read_binary*) and before any use of the BB_XIP field of any BB object.

void create_func_xips()

purpose: To associate a single XIP with each FUNC object. Reads the list of basic blocks which are entry points to this function. Among these, chooses the BB_XIP with the lowest value and fills in the FUNC_XIP with this value.

returns: NONE

input variables: NONE

note(s): The *create_bb_xips*() call **must** be made prior to this call since it uses the BB_XIP of the entry point BBs to create the FUNC_XIP. Since the annotation conventions allow for multiple XIPs to refer to the same function (due to the allowance of multiple entry points to a given function). The creation of the FUNC_XIP cannot be registered. The registration of FUNC_XIP is simply to open up space for the post processor to fill in. Furthermore, only one XIP will be store with the function, that being the entry point with the lowest address. SO, basically what this means is that given any XIP which is recognized as an entry point to a FUNC, the FUNC object will be accessed. However, if the XIP of a FUNC object is request, the entry point XIP with the lowest value will be returned.

Annotation Conventions 6

6.1 Introduction

The Annotation Conventions were developed out of the need to easily combine information from the compiler and from a simulator/performance monitors. Flexibility and extensibility were a primary concern in the design of the interface. The type of information that the compiler and simulator produce is likely to change over time. Using an object/attribute pair based API assures that new information does not break existing application tools.

The following tables list the attributes and object classes chosen by convention, although the use of the library is not limited to the hierarchy described in this section.

6.2 Module Level Static (ANI) Attributes:

Table 6-1. MODULE Attributes (ANI)

Attribute Name	Type	Description
MODULE_NAME	ann_string	Unique per MODULE name of the source file being compiled. Used to identify different file objects
MODULE_POINTERSIZE	ann_uint32	Size of a pointer (in bytes) of the compiled module.
MODULE_INTEGERSIZE	ann_uint32	Size of an integer (in bytes) of the compiled module.
MODULE_LONGSIZE	ann_uint32	Size of a long (in bytes) of the compiled module.
#_MODULE_FILE_LIST	ann_uint32	Number of functions in this object module.
_MODULE_FILE_LIST	ann_obj_list	Array of files in this object module.

6.3 File Level Static (ANI) Attributes:

Table 6-1. FILE Attributes (ANI)

Attribute Name	Type	Description
FILE_NAME	ann_string	Unique per MODULE name of the source file being compiled. Used to identify different file objects
FILE_COMPILATION_DATE	ann_string	Compilation Date of Annotator used.
FILE_COMPILER_VERSION	ann_string	Compilation Version of Annotator used.
FILE_COMPILER_OPTIONS	ann_string	Compiler Options used to compile this module.
#_FILE_FUNC_LIST	ann_uint32	Number of functions in this object file.
_FILE_FUNC_LIST	ann_obj_list	Array of functions in this object file.

6.4 Function Level Static (ANI) Attributes:

Table 6-1. FUNC Attributes (ANI)

Attribute Name	Type	Description
FUNC_NAME	ann_string	Unique per FILE name of the function being compiled. Used to identify different func objects
FUNC_HOME_SOURCE_FILE_NAME	ann_string	Name of the home source file this function came from
#_FUNC_LABEL_LIST	ann_uint32	Number of labels associated with this function
_FUNC_LABEL_LIST	ann_obj_list	Labels associated with this function
#_FUNC_BB_LIST	ann_uint32	# of BB objects belonging to this function
_FUNC_BB_LIST	ann_obj_list	BB objects belonging to this function
FUNC_BB_ENTRY_LIST	ann_uint32_list	BB #’s that represent entry blocks to the function
FUNC_XIP	ann_xip	Lowest address among the entry points to this function..

6.5 Label Level Static (ANI) Attributes:

Table 6-1. LABEL Attributes (ANI)

Attribute Name	Type	Description
LABEL_NAME	string	Unique per FUNCTION name of the label. Used to identify different label objects
#LABEL_BBNUM_LIST	ann_uint32	Number of Basic Blocks which are offset from this label.
LABEL_BBNUM_LIST	ann_uint32_list	Array of basic blocks belonging to this label.
LABEL_XIP	ann_xip	Address of the label.

6.6 Basic Block Level Static (ANI) Attributes

Table 6-1. BB Attributes (ANI)

Attribute Name	type	Description
BB_NUMBER	ann_uint32	Unique per LABEL sequence number of basic block in list
BB_INSTR_SLOT_OFFSET	ann_int32	Offset from the label xip to the bb, expressed in number of instruction slots
BB_INSTR_SLOT_CNT	ann_uint32	Number of slots contained within this basic block
BB_COMPILER_CYCLE_CNT	ann_uint32	Number of static clocks for this basic block as estimated by the compiler
BB_PRED_NUMS	ann_uint32_list	Basic block numbers of predecessors
BB_SUCC_NUMS	ann_uint32_list	Basic block numbers of successors
_BB_INSTR_LIST	ann_obj_list	List of annotated instructions
#_BB_INSTR_LIST	ann_uint32	Number of annotated instructions
BB_XIP	ann_xip	Address of the beginning of the basic block.

6.7 Instruction Level Static (ANI) Attributes:

Table 6-1. INSTR Attributes (ANI)

Attribute Name	type	Description
INSTR_OFFSET	ann_uint16	Unique, per BB, offset in # of instructionslots from beginning of basic block
INSTR_COLUMN	ann_uint8	Source column number
INSTR_SRC_LINE	ann_uint32	Source line number

6.8 Annotations Convention Summary

Although the compiler will not directly annotate FUNC_XIP and BB_XIP (and only indirectly annotate LABEL_XIP) it is necessary to create space for these annotations in the context hierarchy. The annotations post-processing library will handle filling in the information.

6.8.1 Static Attributes And Objects:

MODULE:

(1) MODULE_NAME
(2) MODULE_POINTERSIZE
(3) MODULE_INTEGERSIZE
(4) MODULE_LONGSIZE
(5) _MODULE_FILE_LIST

FILE:

(1) FILE_NAME
(2) FILE_COMPILATION_DATE
(3) FILE_COMPILER_VERSION
(4) FILE_COMPILER_OPTIONS
(5) _FILE_FUNC_LIST

FUNC:

(1) FUNC_NAME
(2) FUNC_HOME_SOURCE_FILE_NAME
(3) FUNC_BB_ENTRY_LIST
(4) _FUNC_LABEL_LIST
(5) _FUNC_BB_LIST
(6) FUNC_XIP*

LABEL:

(1) LABEL_NAME
(2) LABEL_BBNUM_LIST*
(3) LABEL_XIP*

BB:

(1) BB_NUMBER
(2) BB_INSTR_SLOT_OFFSET
(3) BB_INSTR_SLOT_CNT
(4) BB_COMPILER_CYCLE_CNT
(5) BB_PRED_NUMS
(6) BB_SUCC_NUMS
(7) BB_XIP*
(8) _BB_INSTR_LIST

INSTR:

(1) INSTR_OFFSET
(2) INSTR_SRC_COLUMN
(3) INSTR_SRC_LINE

6.9 Annotation Encoding

The advantage of context sensitive annotations is that redundancy, and hence space needed for the annotations in the binary is reduced. The disadvantage is that the annotations consumer has to recreate the entire annotations context to extract a particular annotation, i.e., there is no cheap random access. Since limiting the growth of the size of annotated object files was a goal, we chose to implement a dense, context sensitive annotation encoding scheme.

*** The flexible compiler annotation encodes in the following sections of the assembly file:

Section	Contains Annotations for	Default
.annot_hdr	annotations structure	Always present
.annot	annotations data	Always present

The annotation sections in the object file are marked as readable but “unallocatable” so they do not contribute to the memory foot print of the program.

Examples Of Setting Up And Using Annotations 7

7.1 Using The Annotations API

Annotations represent an object/attribute database. An object/attribute pair uniquely identifies a particular item in the database. All API routines are based on this object and attribute concept. API functions which have an ANO prefix are used to access the static information.

This short tutorial will:

1. Describe how information is stored and consequently accessed.
2. How to set up a simple annotation structure.
3. How to annotate the program.
4. How to write a tool to access the annotated data.

7.2 Class, Object, and Attribute Tokens

Tokens are used throughout the API. Tokens are an efficient way to represent a particular item. Since the API has to be extensible, all data sets are identified by a string. Strings are very costly to deal with in terms of execution time in C programs, so they are replaced by tokens. The API supports four basic token types. The first is called a *class* token. *Class* tokens are used to represent a set of objects. Each *class* has one or more *objects* in it. These *objects* are represented with *object* tokens. An *object* token identifies a particular instance in it's *class*. Since *object* tokens know their *class* in addition to pointing to a particular item, they can be used anywhere a *class* token is required in the API. An *object* can have data associated with it. Each piece of data on an object is called an *attribute*. Each *attribute* on an object has a unique name and is represented by an *attribute* token. All *objects* in a *class* have the same set of *attributes*.

7.3 How are tokens used with annotations?

The compiler currently annotates modules, functions, basic blocks, and instructions. These are considered classes. Let's take functions as an example. This class is called _FILE_FUNC_LIST. Say a program has a procedure called "foobar". Information is stored about it and all the other procedures in the class _FILE_FUNC_LIST. There may be several annotations on "foobar". Each annotation is represented with a separate attribute. For instance, the name of the function is an attribute called FUNC_NAME.

Several routines are provided to create tokens for use in the user's program. Class tokens are made with *ano_get_class_token*. Attribute tokens are created with *ano_get_attr_token*. To make an object token, a class token must first be obtained. Object tokens can be created in one of three ways. A particular object in a class may be picked using *ano_pick_object*. Alternatively, the first object in the class can be obtained using *ano_first_object*. The last method gets an object out of another object.

Subobjects have the unique property that they are both attributes and objects. The object token of the master object and the attribute token of an attribute in the master object are used to get back another object. This is done using *ano_get_object_attr*. Once we have the subobject's token, it can be used to access attribute values in the subobject in the same manner as any other object.

7.4 Walking through the data

Simple API routines are provided to traverse the data in the data base. These routines lend themselves for easy use in a C *for* or *while* loop. To start the traversal, *ano_first_object* should be called. The routine ano_next_object() will give another object on the same class. The next routine does not present the data in any particular order, but it is guaranteed to reach all of the objects in a class while never repeating any object. *ano_next_object* will return an invalid object when all of the object in the class have been covered. This invalid object can be detected with *ano_is_valid_objectp*, hence making this a good test for loop termination.

Traversing the values in an array attribute is done slightly differently. Since the user program must make sure that it does not walk off the end of an array, the size of the array must first be determined. To facilitate this, special attributes are available. These attributes have the same name as an array attribute but are preceded by a # symbol. For instance, if an array of integers is stored in the attribute called VALUES, the attribute that represents the size of the array is called #VALUES. The actual size of the array can be retrieved by using *ano_get_uint32_attr* on the attribute token for #VALUES. Once the size of the array is known, a loop can be written that starts with an index of 0 and continues through one less than the array size. This index is supplied to *ano_get_<type>_attr* (where *<type>* is uint32, flt32, flt64, str, U64, or obj), which will return the appropriate value in the array.

7.5 How to setup a simple annotation structure - A Full Registration Example

A MODULE context has the following attributes:

MODULE_NAME, _MODULE_FILE_LIST

A FILE context has the following attributes:

FILE_NAME, _FILE_FUNC_LIST,

A FUNC context has the following attributes:

FUNC_NAME, _FUNC_BB_LIST, FUNC_XIP

A BB context has the following attributes:

BB_NUMBER, BB_INSTR_SLOT_OFFSET, BB_PRED_NUMS, BB_XIP

The compiler can create such a context tree with the following calls:

```
1. ani_reg_ctxt("MODULE", "ROOT");
```

- Creates a new context container called MODULE which is the child of the ROOT context. Since ROOT is the parent context, MODULE is recognized as the first context in the context tree.

```
2. ani_reg_ctxt(“FILE”, “MODULE”);
```

- Creates a new context container called FILE which is the child of the MODULE context.

```
3. ani_reg_ctxt(“FUNC”, “FILE”);
4. ani_reg_ctxt(“BB”, “FUNC”);
```

- Since the MODULE context is a child of ROOT (see #1), the first object created must be of type MODULE.

```
5. ani_reg_attr(“MODULE_NAME”, “MODULE”);
```

- MODULE_NAME is now an attribute of the MODULE

```
6. ani_reg_attr(“_MODULE_FILE_LIST”, “MODULE”);
```

- Now the data types of MODULE’s attributes can be registered.

```
7. ani_reg_attr_data_type(“MODULE_NAME”, ann_string);
```

- MODULE_NAME refers to a string value.

```
8. ani_reg_attr_data_type(“_MODULE_FILE_LIST”, ann_obj_list);
```

- _MODULE_FILE_LIST is now recognized to refer to a list of objects. MODULE need not have a context identifier registered since ROOT is its parent context (see #1 again) and only one instance of it may be created.

- When the compiler *ano_close_obj*, if the current open object is of type FILE, the FILE object will be closed. The current open object will then become the parent of the FILE object (a MODULE object).

```
9. ani_reg_attr(“FILE_NAME”, “FILE”);
10. ani_reg_attr(“_FILE_FUNC_LIST”, “FILE”);
11. ani_reg_attr_data_type(“FILE_NAME”, ann_string);
12. ani_reg_attr_data_type(“_FILE_FUNC_LIST”, ann_obj_list);
```

- Since _FUNC_LIST refers to a list of objects, it must be of type ann_obj_list.

- Refer to step #20 to see what type of object is registered to be contained in a _FILE_FUNC_LIST.

Now that the FILE context and its associated attributes have been registered, it is time to register where FILE objects will be contained in their MODULE parent and how they may be distinguished from each other in the same list.

```
13. ani_reg_obj_list(“_MODULE_FILE_LIST”, “FILE”, “FILE_NAME”);
```

- This call provides the means for gluing the tree together. In this example, FILE objects are contained in their parent object’s _MODULE_FILE_LIST. They are distinguished from each other by their FILE_NAME value. The registration of the _MODULE_FILE_LIST and FILE_NAME must have occurred prior to this call (calls #6 and #9. Also since FILE_NAME is registered as an identifier of FILE objects in a _MODULE_FILE_LIST, its data type must be registered prior to this call. (call #8) _MODULE_FILE_LIST must now be registered as an object list. (call #13).

- FILE_NAME is registered as an attribute of FILE. When an *ani_open_obj* call is performed by the compiler on FILE_NAME is performed by the compiler, a FILE object will be opened and the object tree will be traversed from the MODULE to the FILE level (Call #2 created the FILE context as a child of MODULE). If an object with the requested FILE_NAME already exists, the object will be reopened else a new one will be created.

```
14. ani_reg_attr(“FUNC_NAME”, “FUNC);
15. ani_reg_attr(“_FUNC_BB_LIST”, “FUNC”);
16. ani_reg_attr(“FUNC_XIP”, “FUNC”);
17. ani_reg_attr_data_type(“FUNC_NAME”, ann_string);
18. ani_reg_attr_data_type(“FUNC_XIP”,ann_xip);
```

```
19. ani_reg_attr_value_type("_FUNC_BB_LIST", ann_obj_list);
```

- Since _FUNC_BB_LIST is a list of objects it must be of type ann_obj_list

```
20. ani_reg_obj_list("_FILE_FUNC_LIST", "FUNC", "FUNC_NAME");
```

- FUNC, a child of FILE (call #3) objects that share same FILE parent will be contained in their parent's _FILE_FUNC_LIST (call #20) The FILE objects in a _FILE_FUNC_LIST may be distinguished from each other by their FUNC_NAME values.

```
21. ani_reg_xiplabel("FUNC_NAME", "FUNC_XIP");
```

- An annotate call to FUNC_NAME will produce a label in the assembly file which will be replaced by the linker with an XIP. When this XIP value is read from the binary it will be stored in the FUNC_XIP field of the same object.

```
22. ani_reg_attr("BB_NUMBER", "BB");
23. ani_reg_attr("BB_INSTR_SLOT_OFFSET", "BB");
24. ani_reg_attr("BB_XIP", "BB");
25. ani_reg_attr_data_type("BB_INSTR_SLOT_OFFSET", ann_uint16);
26. ani_reg_attr_data_type("BB_NUMBER", ann_uint16);
27. ani_reg_attr_data_type("BB_XIP", ann_xip);
28. ani_reg_obj_list("_FUNC_BB_LIST", "BB", "BB_NUMBER");
```

- Objects of type BB are contained in the parent objects _FUNC_BB_LIST field. They may be distinguished from each other in a _BB_LIST by their BB_NUMBER values.

```
29. ani_reg_xipoffset("BB_INSTR_SLOT_OFFSET", "FUNC_XIP", "BB_XIP");
```

- If an annotated BB_INSTR_SLOT_OFFSET is read from the binary, its value will be added to the parent object's FUNC_XIP value and stored in the BB_XIP of the same object.

7.6 How to use the new flexible annotations - A Silly Little Example.

The following is a simple example of using the annotation library (*libani*) given the context hierarchy registered in *Section 7.5 "How to setup a simple annotation structure - A Full Registration Example"* Furthermore it assumes, per the current implementation, that only one object may be open at any one time and traversal of the object tree must be done from parent to child or child to parent (as opposed to child to child or some other arbitrary scheme of object tree traversal).

```
obj_t *cur_open_objp;
```

Initialize the flexible annotation data structures.

```
1. ani_init();
```

Must open objects in order as they appear in the corresponding context hierarchy. Since MODULE is child to ROOT it must be opened first. Note that the ROOT pointer is implied so no object pointer need be passed in.

```
2. cur_open_objp = ani_open_obj("MODULE_NAME", "test.asm", NULL);
```

Now that we have a handle on the object tree through cur_open_objp, open up a new FILE object since it was registered as a child of MODULE. Since the FILE_NAME field of the FILE object was registered as the means of distinguishing different FILE objects, it is necessary to use this attribute to open a FILE object.

```
3. cur_open_objp = ani_open_obj("FILE_NAME", "test.c", cur_open_objp);
```

Open a new FUNC object and a new BB object.

```
4. cur_open_objp = ani_open_obj("FUNC_NAME", "main", cur_open_objp);
```

```
5. cur_open_objp = ani_open_obj(“BB_NUMBER”, 0, cur_open_objp);
```

Annotate an additional attribute of this BB object. Then, since we’re done annotating this BB object close it. Cur_open_objp will now point to the BB’s FUNC parent (identified as the “main” FUNC);

```
6. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 0);
7. cur_open_objp = ani_close_obj(cur_open_objp);
```

Create another BB object and annotate it completely.

```
8. cur_open_objp = ani_open_obj(“BB_NUMBER”, 1, cur_open_objp);
9. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 8);
10. cur_open_objp = ani_close_obj(cur_open_objp);
```

Time to close the FUNC “main” and create a new FUNC “func1” and annotate its BBs.

```
11. cur_open_objp = ani_close_obj(cur_open_objp);
12. cur_open_objp = ani_open_obj(“FUNC_NAME”, “func1”, cur_open_objp);
13. cur_open_objp = ani_open_obj(“BB_NUMBER”, 0);
14. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 0);
15. cur_open_objp = ani_close_obj(cur_open_objp);
16. cur_open_objp = ani_open_obj(“BB_NUMBER”, 4);
17. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 17);
18. cur_open_objp = ani_close_obj(cur_open_objp);

19. cur_open_objp = ani_open_obj(“BB_NUMBER”, 9);
20. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 145);
21. cur_open_objp = ani_close_obj(cur_open_objp);
22. cur_open_objp = ani_close_obj(cur_open_objp);
```

Well, it would appear the BB #1 of FUNC “main” was incorrectly annotated. Time to fix it by annotating the proper BB_INSTR_SLOT_OFFSET. Note that instead of creating a new FUNC object, the currently open FILE’s list of FUNCs will be traversed to see if FUNC “main” exists. If it does, it will be reopened. That same traversal and reopening will occur for BB# 1 of FUNC “main”.

```
23. cur_open_objp = ani_open_obj(“FUNC_NAME”, “main”, cur_open_objp);
24. cur_open_objp = ani_open_obj(“BB_NUMBER”, 1);
25. ani_annot_obj(“BB_INSTR_SLOT_OFFSET”, 15);
26. cur_open_objp = ani_close_obj(cur_open_objp);
27. cur_open_objp = ani_close_obj(cur_open_objp);
28. cur_open_objp = ani_close_obj(cur_open_objp);
29. cur_open_objp = ani_close_obj(cur_open_objp);
```

All done annotating, ready to emit the annotations into the assembly file.

```
30. ani_close(ASM_FP);
```

Sample Code for a simple ANO based tool 8

8.1 The *bb_count* tool

This section shows a sample tool that uses the annotations within a binary. The tool shown counts the total number of basic blocks in the binary. It counts the number of basic blocks in each function and the total number of instructions in each function.

Data is stored as attribute-value pairs within objects in the binary. The first step is to access these objects and to then query the values of the attributes within the objects. To perform these queries, tokens need to be acquired. For example, to access the functions within the binary, one needs to first acquire a function class *object_token*. This is done by making a call to *ano_get_class_token*. This returns an uninstantiated token to the desired object. The actual objects are acquired by calling *ano_first_object/ano_next_object*.

Once a valid object is acquired, queries can be made on the attributes corresponding to that object. To do this, one needs to acquire an *attr_token* that refers to the desired attribute. This is done by calling *ano_get_attr_token* with the corresponding string as defined in the ANO dictionary. With a valid *object_token* and *attr_token*, the value corresponding to the attribute can be obtained by calling *ano_get_<type>_attr*, where *<type>* is the type of the value (uint32, flt32, str. etc.) corresponding to the attribute. The sample program below shows the interaction of the objects, attributes and tokens to perform a simple task.

```
/*
 * A sample program that counts the basic blocks in a binary
 *
 * Libraries needed: -lano -lannotate -lmofl -liel
 */
#include <stdio.h>

/* These includes are required for using the ANO API */
#include “ano/ano_api.h”

/* Include the post processing layer */
#include “anopp/anopp.h”



char static_file_to_parse[1000] = “a.out”;/* default file to read */


void main (int argc, char **argv)
{
    object_token ot_func_d;
    object_token *ot_func = &ot_func_d;

    object_token ot_bb_d;
    object_token *ot_bb = &ot_bb_d;

    object_token ot_instr_d;
    object_token *ot_instr = &ot_instr_d;
```

```
attr_token at_fname;
attr_token at_nbbs;
attr_token at_bb;
attr_token at_bbnum;
attr_token at_ninstrs;
attr_token at_ano_xip;
int total_instructions = 0;/* running total */

/* read annotated binary*/
printf ("ano_read_binary %s\n", static_file_to_parse);
if (ano_read_binary (static_file_to_parse)) {
   fprintf (stderr, "ano_read_binary failed\n");
   exit (1);
}

/* Call through the ANO post processing layer to create the BB XIPs */
create_bb_xips();
create_func_xips();

/* this is a "fetch an empty function class token" operation */
ano_get_class_token ("_FUNC_LIST", ot_func);

/* initialize static attribute tokens */
at_fname      = ano_get_attr_token ("FUNC_NAME");
at_nbbs       = ano_get_attr_token ("#_FUNC_BB_LIST");
at_bb         = ano_get_attr_token ("_FUNC_BB_LIST");
at_bbnum      = ano_get_attr_token ("BB_NUMBER");
at_ninstrs    = ano_get_attr_token ("BB_INSTR_SLOT_CNT");
at_ano_xip    = ano_get_attr_token ("BB_XIP");

/* instantiate ot_func to refer to the first function in the static list */
for (ano_first_object (ot_func); /

   /* test if we have a good function token or we are done with the loop */
   ano_is_valid_objectp (ot_func);

   /* instantiate ot_func to refer to the next function in the list */
   ano_next_object (ot_func)/) {

char *fname;
int bb_index, nbbs;
int valid;
int function_instructions = 0;

fname = ano_get_str_attr (ot_func, at_fname, &valid);
nbbs = ano_get_uint32_attr (ot_func, at_nbbs, &valid);

for (bb_index = 0; bb_index < nbbs; bb_index ++) {
    int ninstrs, bbnum;
    U64 xip_64;

   /* this returns an instantiated bb object, which is
      considered an attribute of a function */
    ano_get_object_array_attr (ot_func, at_bb, bb_index, ot_bb, &valid);

    bbnum = ano_get_uint32_attr (ot_bb, at_bbnum, &valid);
    ninstrs = ano_get_uint32_attr (ot_bb, at_ninstrs, &valid);

    xip_64 = ano_get_U64_attr (ot_bb, at_ano_xip, &valid);
```

```
        if (valid)
        {
            printf (“function:  %20s  bb: %3d  XIP: 0x%08x%08x\n”,
                fname, bbnum, ANO_HIGH(xip_64), ANO_LOW(xip_64));

            function_instructions += ninstrs;
            total_instructions += ninstrs;
        }
    }

    printf (“function: %20s, #bbs: %d, #instructions: %d\n”, fname, nbbs,
function_instructions);
    }

    printf (“total_instructions:  %d\n”, total_instructions);
}
```